



## **Highlander Host Software Kernel Manager API Definition**

© COPYRIGHT VideoLogic Limited, 1996.

The content of this document is proprietary to VideoLogic limited. It may not be copied or its contents disclosed without the prior consent of the appropriate VideoLogic director.

This document is made available to NEC for the purposes of Highlander joint development only. It may be copied or its contents disclosed only to NEC personnel who require such disclosure for work on the above project. It must not be disclosed to NEC personnel involved in any projects which may constitute a competing activity in any way.

Document ID : guid003

Issue Number : Draft A

Issue Date : 24th June 1996

Author : John Russell



## DOCUMENT HISTORY

Issue	Date	Changes / Comments
A	24 <sup>th</sup> June 1996	First Issue, some API's yet to be defined, others may change.
B	22 <sup>nd</sup> July 1998	Everything changed



## TABLE OF CONTENTS

1. Introduction.....	1
2. What the Kernel Manager Is .....	1
3. Building The Kernel Manager .....	1
4. Communicating with the Kernel Manager.....	2
5. Basic Concepts.....	2
5.1 Connecting with a device.....	2
5.2 Errors.....	2
5.3 Threads And Code Loading .....	2
5.4 Symbols .....	3
5.5 Memory Allocation.....	3
5.5.1 Frame Buffer Allocation.....	3
5.5.2 Host Allocation .....	4
5.6 Asynchronous and Synchronous commands .....	4
5.6.1 Asynchronous commands .....	4
5.6.2 Synchronous commands .....	5
5.7 Interrupts.....	7
5.8 State And Power Mode Changes .....	7
6. Ring0 Function & Macro Reference.....	8
6.1.1 KMLoadCodeFragment .....	8
6.1.2 KMUnloadCodeFragment.....	9
6.1.3 KMFindThreadWithCaps .....	9
6.1.4 KMGetCodeCtlBlkPtr .....	10
6.1.5 KMGetSymbolHostAddress .....	11
6.1.6 KMGetSymbolHighAddress.....	11
6.1.7 KMMMapOSDevCookieToDevice.....	11
6.1.8 KMAllocStaticFBMem.....	11
6.1.9 KMAllocMovFBMem .....	12
6.1.10 KMFreeStaticFBMem .....	13
6.1.11 KMFreeMovFBMem .....	13
6.1.12 KMLockMoveableBuffer .....	14
6.1.13 KMUnlockMoveableBuffer .....	14



6.1.14 KMGetAvailMemory .....	14
6.1.15 KMMapLinToPhys .....	16
6.1.16 KMAllocComIF .....	16
6.1.17 KMFreeComIF .....	16
6.1.18 KMInstallVxDIntHandler .....	16
6.1.19 KMRemoveVxDIntHandler .....	16
6.1.20 KMAllocPages .....	17
6.1.21 KMFreePages .....	17
6.1.22 KMGetCmndBufferInfo .....	17
6.1.23 KMSetCmndBufferInfo .....	18
6.1.24 KMAllocHostMem .....	18
6.1.25 KMFreeHostMem .....	19
6.1.26 KMWriteOBC .....	19
6.1.27 KMReadOBC .....	19
6.1.28 KMWriteOBCCI .....	20
6.1.29 KMReadOBCCI .....	20
6.1.30 KMReadOBCNB .....	20
6.1.31 KMGetCardInfo .....	21
6.1.32 KMAllocHostCacheCoherentHeap .....	22
6.1.33 KMFreeHostCacheCoherentHeap .....	23
6.1.34 KMPower .....	23
6.1.35 KMCorePLL .....	23
6.1.36 KMState .....	24
6.1.37 KMGetAPITable .....	24
6.1.38 KMRegisterStateChangeCallback .....	25
6.1.39 KMDeRegisterStateChangeCallback .....	25
7. DeviceIOControlInterface reference .....	25
7.1.1 KM_API_LOAD_CODEFRAGMENT .....	26
7.1.2 KM_API_UNLOAD_CODEFRAGMENT .....	26
7.1.3 KM_API_FIND_THREAD_WITH_CAPS .....	26
7.1.4 KM_API_GET_CODECTLBLOCKPTR .....	27
7.1.5 KM_API_GET_SYMBOL_ADDRESS .....	27



---

7.1.6 KM_API_GET_API_TABLE.....	27
7.1.7 KM_API_MAP_OSDEVCOOKIE_TO_DEVICE.....	28
7.1.8 KM_API_GET_CARD_INFO .....	28
7.1.9 KM_API_LOCK_MOVEABLE_BUFFER .....	28
7.1.10 KM_API_UNLOCK_MOVEABLE_BUFFER.....	29
7.1.11 KM_API_MAP_LIN_TO_PHYS.....	29
7.1.12 KM_API_ALLOC_STATIC_FB_MEMORY .....	29
7.1.13 KM_API_FREE_STATIC_FB_MEMORY .....	30
7.1.14 KM_API_GET_AVAILABLE_MEMORY .....	30
7.1.15 KM_API_ALLOC_MOV_FB_MEMORY .....	30
7.1.16 KM_API_FREE_MOV_FB_MEMORY.....	31
7.1.17 KM_API_ALLOC_HOST_MEMORY .....	31
7.1.18 KM_API_FREE_HOST_MEMORY .....	32
7.1.19 KM_GET_SERVICE_TABLE.....	32
7.1.20 KM_API_ALLOC_COMIF.....	32
7.1.21 KM_API_FREE_COMIF .....	33
7.1.22 KM_API_SET_CMND_BUF_INFO .....	33
7.1.23 KM_API_GET_CMND_BUF_INFO.....	33
8. Macros.....	34
8.1.1 FENCED_WRITE( dwAddr, dwData ) .....	34
8.1.2 CMND_BUFFER_SIZE .....	34
8.1.3 CMND_BUFFER_OVERFLOW .....	34
8.1.4 GET_PCCB_SPACE(WOff,ROff).....	34
8.1.5 UPDATE_PCCB_WOFF(USize,WOff).....	34
8.1.6 CalcParamSize .....	34
8.1.7 ReleaseCCB .....	34
8.1.8 GetCCB.....	35





## 1. INTRODUCTION

The Kernel Manager is one of the low level system components comprising the PMX driver. It is intended to provide a mostly operating system independent layer with which to access various features of the hardware. It is responsible for device initialisation, memory allocation, interrupt handling, and power management. This document defines the interface to the various functions provided by the Kernel manager.

## 2. WHAT THE KERNEL MANAGER IS

The Kernel Manager is a low level system component. This means that it runs as a VxD in Win9X, and as a Kernel-mode driver in Windows NT™.

## 3. BUILDING THE KERNEL MANAGER

You can build the Kernel Manager from either the nt4 or the win9x directory using the respective make files. Various options are supported:

DEBUG=1 does a debug build

NO\_HOST\_CBUFF=1 puts the CCB on the host, rather than in AGP memory

SIM=1 does a build suitable for running simulations with simstim. Simulates a single PMX1C.

SIM\_S3=1 does a build suitable for running the PMX software emulated drivers on an S3 card.

PMXLC=1 if used in combination with SIM=1 makes the simulated device a PMXLC rather than a PMX1C. It is now used to distinguish a PMXLC/PMX2 build from a PMX1C build.

On a SIM=1 build, the Kernel Manager simulates a device with 4Mbytes of frame buffer, and no texture heap. By creating a registry key in HKEY\_LOCAL\_MACHINE called PMXSIM, you can vary the amount of texture heap. Simply set the entry SimTextureMemorySize to the size (in hex) of the texture heap you wish to simulate.

You can also set the simulated physical frame buffer base by setting the the entry SimPhysicalBase, again in PMXSIM.



## 4. COMMUNICATING WITH THE KERNEL MANAGER

The Kernel manager provides two interfaces to the same set of functions, to enable it to be called from both ring 3 (not supported on NT) and ring 0. The ring 3 interface allows access using a DeviceIOControl call. The ring 0 interface is accessed by calling a VxD service (on Win9X), or an IOCTL (on NT), which returns a table of function pointers, which you can then call as you desire.

The declaration of the Kernel Manager interface functions and #defines is available in the file kmapi.h.

## 5. BASIC CONCEPTS

### 5.1 Connecting with a device

The first thing you need to do is to get the Kernel Manager's function table. This is accomplished by :

on Win9X : call the DIOCGetServiceTable function

on WINNT : issue a KM\_IOCTL\_GET\_SERVICE\_TABLE IOCTL.

These will return a pointer to a table of functions (of type KM\_SERVICE\_TABLE), which can then be called directly (from ring 0).

You can use the KMMapOSDevCookieToDevice function to obtain a PDEVICE\_CTLBLK given a windows DEV\_NODE. Use dnOsDevCookie = 1 to get at a simulated device. The returned PDEVICE\_CTLBLK acts as a handle to a particular PMX device. From the client's point of view it acts purely as a unique identifier for a device, with no internal structure. If you are debugging through the Kernel Manager, the internal structure of a PDEVICE\_CTLBLK can be accessed by including kminc.h instead of kmapi.h.

### 5.2 Errors

Most Kernel Manager functions return an error code. You should test this after every call and take appropriate action if a failure code is returned. Kmapi.h defines a large set of error codes, but the most important one is KM\_OK, which indicates success.

### 5.3 Threads And Code Loading

The PMX primary core supports several threads (the exact number varies depending on the chip revision). Each of these threads has different hardware capabilities. You can find what thread has what capabilities by calling KMFindThreadWithCaps, setting the bits in the dwCaps field of the FIND\_THREAD structure according to the capabilities you desire, and the device you want to interrogate.

Typical capabilities are:



```
def KM_HCAPS_BG
```

```
KM_HCAPS_DACCORE
```

The function will set the dwThreadNum field of the structure to be a unique identifier for a thread possessing the desired capabilities. If no valid thread can be found, the function will return an error code != KM\_OK.

Once you've acquired a thread number, you can load code onto that thread. This is simply achieved by calling KMLoadCodeFragment, setting the psDeviceCtlBlk field to the unique identifier for the PMX device you require, and the dwThread field to the number returned from KMFindThreadWithCaps. You also need to setup the pdwInfoBuffer field to point to an area of memory holding the binary image of a .pce file containing the code fragment you want to load.

## 5.4 Symbols

A primary core code fragment can export symbol addresses to the external world by using the assembler directive EXPORT <label>. When the code is loaded, the Kernel Manager records the address of this label in a table, which can be queried using the KMGetSymbolAddress function. Both linear (host) and physical addresses can be obtained for any exported symbol.

A code fragment may also contain a table called an API table within it. This is just an array of exported symbols. This table can be retrieved after a fragment has been loaded by calling the KMGetAPITable function. The table contains the physical addresses of the corresponding labels in the code after it has been loaded.

## 5.5 Memory Allocation

The Kernel Manager provides functions to allocate memory from the host's shared arena, and from any card's frame buffer.

### 5.5.1 Frame Buffer Allocation

The Kernel Manager creates several distinct heaps in the frame buffer at initialization time. Any of these heaps can be allocated from using either a static or a moveable allocation. A static allocated block has its location fixed for as long as it exists. A moveable allocated block can be moved by the memory manager whenever it wants to reduce fragmentation. Moveable blocks can be locked to prevent the memory manager from moving them.

The heaps created in the frame buffer are :-



KM\_FB\_GENERAL\_HEAP -- holds primary core code, primary surface etc.  
KM\_FB\_PROTECT\_HEAP -- holds the asynchronous command interfaces (see below)  
KM\_FB\_ICACHE2\_HEAP -- holds real-time code (1K maximum)  
KM\_FB\_TEXTURE\_HEAP -- holds 3D textures

Allocations can be made on any alignment that is a power of two greater than or equal to 8. Similarly you can choose not to split small allocations across page boundaries by setting the KM\_HEAP\_NOT\_SPLIT\_PAGE flag. See the reference section for KMAAllocStaticFBMem and KMAAllocMoveableFBMem for details.

### 5.5.2 Host Allocation

All host allocations are satisfied from shared memory. You can allocate from one of three heaps using the single function KMAAllocHostMem

KM\_HOST\_PAGEABLE\_HEAP

KM\_HOST\_NON\_PAGEABLE\_HEAP

KM\_HOST\_NON\_CACHED\_HEAP (turns off caching by writing to the relevant PTE's in the processor. Access to this memory is very slow – you are better off using AGP GART memory).

There is also an API for allocating memory which is cache coherent with the primary core, see

KMAAllocHostCacheCoherentHeap for details.

## 5.6 Asynchronous and Synchronous commands

You can call routines on the primary core using a mechanism similar to a remote procedure call. This is accomplished either by writing into the circular command buffer, or by using an asynchronous command interface.

### 5.6.1 Asynchronous commands

In order to use an asynchronous command, you first call KMAAllocComIF to allocate a structure called a command interface. This contains a dwExeAddr which is (physical) address of the primary core routine to execute. It also contains an array of eight DWORDs,

adwUserData, which hold data to be passed to the asynchronous command. This data is accessible using the PARAMS register in the primary core code.

Writing into these fields must be ordered so that adwUserData is filled out before dwExeAddr (a FENCED\_WRITE macro provided in kmapi.h is suitable for enforcing this ordering).

Having filled out these fields, you must then perform

```
*(psCommandInterface->pdwKickerAddr) = 0x8001
```

This informs the hardware that it has a new command to execute, by writing to the relevant 'kicker register' for the thread.

The command interface can be reused to send more than one command, but you must wait until the last command is complete before sending a second command with the same command interface.

Asynchronous commands execute as soon as possible, and may execute out of order.

The commands interface can be freed after use by calling KMFreeComIF.

### 5.6.2 Synchronous commands

Most commands to the primary core (blts for example) are synchronous commands. All synchronous commands are guaranteed to execute in strict sequence (although an asynchronous command can interrupt this sequence at any time). A synchronous command is called by writing a command packet into a cyclic buffer called the circular command buffer.

### 5.6.3 The circular buffer.

This is a region of memory with 64K of active data space, and a 4K overflow buffer. Two global cursors into this region are maintained, a read Offset (Roff) and a write offset (Woff). These cursors live on the cache coherent region of the host (although the primary core will keep a local copy of the read offset, and update the host version now and again to maintain coherence). The primary core iterates around the CCB, calling the commands embedded therein in sequence.

Command packets contain a primary core exe address field and a command size field (measured in bytes), at the head of their data. Because of certain hardware limitations, a command packet must be a multiple of 64 bytes in size.

The sequence for performing a writing a command into the circular command buffer is:

- 1) Exchange 1 with the dwAccess field of the circular command buffer



- 2) repeat until the result of the exchange is 0
- 3) Call GET\_PCCCB\_SPACE (a macro in kmapi.h)
- 4) If there is insufficient space then wait or split the command
- 5) If your command may be > 4K then you must test to make sure that Woff + CommandSize < 68K. If its not then you must split the command into 4K chunks. (or you could write a dummy command to wrap around etc.)
- 6) Write the command into the buffer at the Write offset.
- 7) Call the UPDATE\_PCCCB\_WOFF(CommandSize) macro (which does Woff = (Woff + CommandSize) & 65535)
- 8) Hit the kicker register for the BG0 thread. (write a 1 into it)
- 9) Write a zero into the dwAccess field of the circular command buffer

Commands may be agglomerated together by writing them all at once, but you must remember to make sure that they meet the 4K condition 5). You must also write the number of agglomerated commands into the kicker register, rather than just a 1.

There are two special commands:

NoCmd:

NO-OP command handler. The size field is used to tell the primary core how much to update its read pointer by.

Status:

This allows us to determine the completion status of the preceding commands. It takes three parameters: the address to write to and the value to write when the preceeding operations complete. It also takes a flag indicating whether or not we should flush any pipelines after the blt. [You'd have to set this if you are using the status command as a synchronization point rather than indicating the ability to reuse a cache entry].

You can get obtain the dwAccess, Roff, and Woff pointers, as well as the linear base of the CCB by calling KMGetCmndBufferInfo. The Roff and Woff pointers can be set by calling KMSetCmndBufferInfo.



## 5.7 Interrupts

tbd

## 5.8 State And Power Mode Changes

A state change occurs when a particular device goes into or out of a full-screen DOS mode, or changes its power mode.

You can register a callback function with the KM. This will be called both pre and post mode change & power down.

The callback function should be of the form:

```
Fn( PDEVICE_CTLBLK psDeviceCtlBlk,  DWORD dwStateFlags,  DWORD
dwContextLossSeverityFlags )
```

where dwStateFlags indicates the type of the message :-

PMX\_PRE\_WINDOWS\_MODE\_CHANGE

PMX\_PRE\_HIREZ\_TO\_VGA

PMX\_POST\_HIREZ\_TO\_VGA

PMX\_PRE\_VGA\_TO\_HIREZ

PMX\_POST\_VGA\_TO\_HIREZ

PMX\_PRE\_XY\_REZ\_CHANGE

PMX\_POST\_XY\_REZ\_CHANGE

PMX\_PRE\_Z\_REZ\_CHANGE

PMX\_POST\_Z\_REZ\_CHANGE

PMX\_PRE\_POW\_STATE0

PMX\_POST\_POW\_STATE0

PMX\_PRE\_POW\_STATE1

PMX\_POST\_POW\_STATE1



```
PMX_PRE_POW_STATE2
PMX_POST_POW_STATE2
PMX_PRE_POW_STATE3
PMX_POST_POW_STATE3
```

and `dwContextLossSeverityFlags` is currently one of:

```
PMX_NO_CONTEXT_LOSS
PMX_SEVERE_LOSS_OF_CONTEXT
```

(the callback function has a typedef to `STATE_CALLBACK` in `kmapi.h`)

On getting the "PRE" notification, (which can be conveniently tested by anding with the `PMX_PRE_STATE_CHANGE_MASK`, defined in `kmapi.h`), you should wait for all of your asynchronous commands that depend on the display mode to complete before returning.

In the event of getting a `PMX_SEVERE_LOSS_OF_CONTEXT` message at the start of a state change, you should preserve all your context somewhere. (for later restoring).

## 6. RING0 FUNCTION & MACRO REFERENCE

Note that many of the structures described here contain a `dwStructSize` field. As far as I'm aware, this is never used in the Kernel Manager, and consequently contains no valid information. It is a hangover from earlier incarnations of the system.

### 6.1.1 KMLoadCodeFragment

```
DWORD __stdcall KMLoadCodeFragment(PCODE_DETAILS);
```

This function loads primary core code onto the given thread on the given device. It takes a pointer to the following structure, and fills out the `psCodeCtlBlk` field with a handle to the loaded thread.

```
typedef struct _CODE_DETAILS_
{
```



```

} CODE_DETAILS, *PCODE_DETAILS;

```

DWORD	dwStructSize;	(Do not use)
DWORD	dwErrorCode;	(Output)
PDWORD	pdwInfoBuffer;	(Input)
DWORD	dwLoadFlags;	(Set this to 0)
DWORD	dwThread;	(Input)
PCODE_CTLBLK	psCodeCtlBlk;	(Output)
PDEVICE_CTLBLK	psDeviceCtlBlk;	(Input)

```

} CODE_DETAILS, *PCODE_DETAILS;

```

psDeviceCtlBlk is the device handle returned by KMMMapOSDevCookieToDevice.

psCodeCtlBlk is a unique identifier for the code fragment in the frame buffer. It is passed to KMGetSymbolHostAddress to indicate which fragment we're looking for symbols within.

### 6.1.2 KMUnloadCodeFragment

```

DWORD __stdcall KMUnloadCodeFragment(PCODE_CTLBLK psCodeCtlBlk);

```

Frees the the code fragment identified by psCodeCtlBlk. Deallocates all frame buffer and host memory that was allocated for it.

### 6.1.3 KMFindThreadWithCaps

```

DWORD __stdcall KMFindThreadWithCaps(PFIND_THREAD);

```

This locates a thread on a specified device having the given capabilities. It uses the following structure:

```

typedef struct _FIND_THREAD_

```

```

{

```

DWORD	dwStructSize;	(Input)
DWORD	dwCaps;	(Input)
DWORD	dwThreadNum;	(Output)
PDEVICE_CTLBLK	psDeviceCtlBlk;	(Input)
DWORD	dwErrorCode;	(Output)



```
} FIND_THREAD, *PFIND_THREAD;
```

You must set up the capabilities request flags by orring together the following macros:

```
KM_HCAPS_BG          -- Thread is a background thread
KM_HCAPS_2DCORE      -- Thread can use a 2D core
KM_HCAPS_RT          -- Thread is real-time (obsolete)
KM_HCAPS_KERNEL
KM_HCAPS_HARDRT      -- Thread is hard real time (obsolete)
KM_HCAPS_DACCORE     -- Thread can use a DAC core
KM_HCAPS_VOUTCORE    -- Thread can use a video out core
KM_HCAPS_VINCORE     -- Thread can use a video in core
```

The function fills out the dwThreadNum with the number of a thread on the specified device having the desired capabilities. This number is usually passed to KMLoadCodeFragment to indicate which thread to load code onto.

#### 6.1.4 KMGetCodeCtlBlkPtr

```
DWORD __stdcall KMGetCodeCtlBlkPtr(PGET_CTLBLKPTR);
```

This function allows you to obtain the unique identifier for a loaded primary core code fragment by specifying which device it exists on, and what its name is.

```
typedef struct _GET_CTLBLKPTR_
{
    DWORD          dwStructSize;      (Input)
    DWORD          dwErrorCode;       (Output)
    PSZ            pszFragName;       (Input)
    PDEVICE_CTLBLK psDeviceCtlBlk;    (Input)
    PCODE_CTLBLK   psCodeCtl;         (Output)
} GET_CTLBLKPTR, *PGET_CTLBLKPTR;
```



On return, the psCodeCtl field is filled out with the handle for the requested fragment. If the system cannot find the requested code fragment, then dwErrorCode will be set to KM\_ERROR\_SYMBOL\_NOT\_FOUND, otherwise it will be set to KM\_OK.

### 6.1.5 KMGetSymbolHostAddress

```
DWORD (__stdcall *KMGetSymbolHostAddress)( PSZ    pszName,    PCODE_CTLBLK
psCodeCtl, PVOID *ppvAddress );
```

Given the handle to the code fragment, and the name of a symbol, this function fills out the ppvAddress parameter with the linear (host) address of the requested symbol. If the system cannot find the requested symbol, then KM\_ERROR\_SYMBOL\_NOT\_FOUND will be returned, otherwise KM\_OK will be returned.

### 6.1.6 KMGetSymbolHighAddress

```
DWORD __stdcall KMGetSymbolHighAddress( PSZ    pszName,    PCODE_CTLBLK
psCodeCtl, PDWORD pdwHighAddress );
```

Given the handle to the code fragment, and the name of a symbol, this function fills out the ppvAddress parameter with the physical address of the requested symbol. If the system cannot find the requested symbol, then KM\_ERROR\_SYMBOL\_NOT\_FOUND will be returned, otherwise KM\_OK will be returned.

### 6.1.7 KMMMapOSDevCookieToDevice

```
DWORD __stdcall KMMMapOSDevCookieToDevice(DWORD    dnOSDevCookie,
PDEVICE_CTLBLK * ppsDCB);
```

This function translates between the operating systems' unique device handle and a device handle that the Kernel Manager understands (A PDEVICE\_CTLBLK). As a special case, passing in dnOSDevCookie = 1 will, when the Kernel Manager has been built with a SIM=1 flag, return a handle for the simulated device.

### 6.1.8 KMAAllocStaticFBMem

```
DWORD KMAAllocStaticFBMem( PDEVICE_CTLBLK psDeviceCtlBlk, DWORD dwFlags,
DWORD dwSize, DWORD dwAlignment, PVOID *ppvLinAddr, PDWORD pdwPhysAddr
);
```



Allocates a static block in the given devices' frame buffer. A static block cannot be moved by the memory manager.

psDeviceCtlBlk is the handle of the device whose frame buffer is to be allocated from

dwFlags can be one of

KM\_FB\_GENERAL\_HEAP

KM\_FB\_PROTECT\_HEAP

KM\_FB\_ICACHE2\_HEAP

KM\_FB\_TEXTURE\_HEAP

and any of these can be orred with

KM\_HEAP\_NOT\_SPLIT\_PAGE (setting this flag will prevent a block being allocated across a page boundary).

dwAlignment is the byte - alignment to allocate the block on. This must be greater than or equal to 8, and also must be a power of two.

ppvLinAddr is a pointer to a PVOID which receives the linear address of the allocated block

pdwPhysAddr is a pointer to a DWORD which receives the physical address of the allocated block.

### 6.1.9 KMAAllocMovFBMem

DWORD \_\_stdcall KMAAllocMovFBMem(PDEVICE\_CTLBLK psDeviceCtlBlk, DWORD dwFlags, DWORD dwSize, DWORD dwAlignment, PVOID \*ppvHandle);

Allocates a moveable memory block in the given devices frame buffer. An (unlocked) moveable block may be moved by the memory manager at any time. Such blocks may be locked and unlocked by calling KMLockMoveableBuffer and KMUnlockMoveableBuffer.

Locking a block will prevent its relocation by the memory manager.

psDeviceCtlBlk is the handle of the device whose frame buffer is to be allocated from

dwFlags can be one of

KM\_FB\_GENERAL\_HEAP

KM\_FB\_PROTECT\_HEAP

KM\_FB\_ICACHE2\_HEAP

KM\_FB\_TEXTURE\_HEAP

and any of these can be orred with



KM\_HEAP\_NOT\_SPLIT\_PAGE (setting this flag will prevent a block being allocated across a page boundary).

dwAlignment is the byte - alignment to allocate the block on. This must be greater than or equal to 8, and also must be a power of two.

ppvHandle is a pointer to be filled out with a handle to the allocated block. Its only valid uses are to be passed to KMLockMoveableBuffer or KMFreeMovFBMem.

#### 6.1.10 KMFreeStaticFBMem

```
DWORD __stdcall KMFreeStaticFBMem(PDEVICE_CTLBLK psDeviceCtlBlk, DWORD dwFlags, PVOID pvLinAddr);
```

Frees a block that was allocated by KMAllocStaticFBMem.

psDeviceCtlBlk is the handle for the device in which the block was allocated.

dwFlags indicates which heap the block was allocated in. i.e. one of:

KM\_FB\_GENERAL\_HEAP

KM\_FB\_PROTECT\_HEAP

KM\_FB\_ICACHE2\_HEAP

KM\_FB\_TEXTURE\_HEAP

pvLinAddr points to the block to be freed.

#### 6.1.11 KMFreeMovFBMem

```
DWORD (__stdcall *KMFreeMovFBMem)(PDEVICE_CTLBLK psDeviceCtlBlk, DWORD dwFlags, PVOID pvMemHandle);
```

Frees a block that was allocated by KMAllocMovFBMem.

psDeviceCtlBlk is the handle for the device in which the block was allocated.

dwFlags indicates which heap the block was allocated in. i.e. one of:

KM\_FB\_GENERAL\_HEAP

KM\_FB\_PROTECT\_HEAP

KM\_FB\_ICACHE2\_HEAP

KM\_FB\_TEXTURE\_HEAP



pvMemHandle is the handle to the block to be freed.

### 6.1.12 KMLockMoveableBuffer

DWORD \_\_stdcall KMLockMoveableBuffer(PVOID pvMemHandle, PVOID \*ppvLinAddr, PDWORD pdwPhysAddr);

Locks a moveable block so it can't be moved.

pvMemHandle is the the handle for the block to be locked.

ppvLinAddr - pointer to be filled out with the linear address of the locked block.

pdwPhysAddr - pointer to be filled out with the physical address of the locked block.

Note that the two returned pointers only remain valid so long as the block is locked.

### 6.1.13 KMUnlockMoveableBuffer

DWORD \_\_stdcall KMUnlockMoveableBuffer(PVOID pvMemHandle);

Unlocks the moveable block with the given handle.

### 6.1.14 KMGetAvailMemory

DWORD \_\_stdcall KMGetAvailMemory(PAVAIL\_MEM);

This queries the given device for the amount of free space it contains in a given heap or the maximum free space for full-screen DOS modes.

Heaps are arranged as :

Top of heap

Static portion

Shared Area

Moveable portion

Bottom of heap

So the static heap grows downwards, and the moveable heap grows upwards.



```
typedef struct _AVAIL_MEM_
```

```
{
    DWORD    dwStructSize;           (Input)
    DWORD    dwErrorCode;            (Output)
    DWORD    dwFlags;                (Input)
    DWORD    dwModeSpace;            (Output)
    DWORD    dwFreeInMoveable;       (Output)
    DWORD    dwLargestInMoveable;    (Output)
    DWORD    dwLargestInStatic;      (Output)
    DWORD    dwFreeInStatic;         (Output)
    DWORD    dwCard;                 (Output)
    PDEVICE_CTLBLK psDeviceCtlBlk;   (Input)
} AVAIL_MEM, *PAVAIL_MEM;
```

dwStructSize – not used

dwErrorCode – not used

dwFlags – which heap to scan for free space (will usually be KM\_FB\_GENERAL\_HEAP). You can indicate (by orring in):-

KM\_HEAP\_MODE\_VALIDATION – set only dwModeSpace, without bothering to calculate the other heap parameters.

KM\_HEAP\_COMPACT – perform a heap compaction before measuring the available space (very slow).

dwModeSpace – number of bytes from the bottom of the specified heap to the bottom of the static portion of that heap.

dwFreeInMoveable – total amount of free space in the moveable portion of the heap.

dwLargestInMoveable – largest block in the moveable portion of the heap. This is the largest allocation size that can be guaranteed to succeed. (Although larger allocations may succeed if the memory manager manages to compact the memory well).

dwLargestInStatic – as above, but this gives information on the static portion of the heap.

dwFreeInStatic – as above, but this gives information on the static portion of the heap.

dwCard – total SDRAM on the card (in bytes)

### 6.1.15 KMMMapLinToPhys

```
DWORD __stdcall *KMMMapLinToPhys(PVOID);
```

This is a utility service which simply calculates the physical address corresponding to a given linear address.

### 6.1.16 KMAllocComIF

```
DWORD __stdcall KMAllocComIF( PDEVICE_CTLBLK psDeviceCtlBlk, DWORD
dwThread, DWORD dwFlags, PCOMMANDIF *ppsCommandIF, PDWORD
pdwHPtrCommandIF );
```

Allocates a structure called a Command Interface which is used for sending asynchronous commands to the primary core.

psDeviceCtlBlk – handle for the device on which to create the command interface

dwThread – thread number on which to create the command interface

dwFlags – should be 0, or KM\_COMIF\_GETFIRST to get at the first command interface (which is otherwise reserved). For non-background threads, the reserved command interface is the only one supported. This means that for non-background threads, you MUST set KM\_COMIF\_GETFIRST.

ppsCommandIF – pointer to fill out with the address of the allocated command interface

pdwHPtrCommandIF – pointer to a DWORD to fill out with the physical address of the allocated command interface.

### 6.1.17 KMFreeComIF

```
DWORD __stdcall KMFreeComIF(PCOMMANDIF psCommandIF);
```

### 6.1.18 KMInstallVxDIntHandler

```
DWORD __stdcall KMInstallVxDIntHandler(PDEVICE_CTLBLK psDCB,
PINT_CALLBACK pfnCallback, DWORD dwContext, DWORD dwIntMask);
```

### 6.1.19 KMRemoveVxDIntHandler

```
DWORD __stdcall KMRemoveVxDIntHandler(PDEVICE_CTLBLK psDCB, DWORD
dwIntMask);
```

### 6.1.20 KMAllocPages

PVOID \_\_stdcall KMAllocPages(DWORD dwNumPages, DWORD dwUserFlags);

Allocates the required number of pages on the host PC, starting on a page boundary.

dwUserFlags can be one of:

GHEAP\_BLOCK\_LOCK for locked (unpageable) pages

GHEAP\_BLOCK\_UNCACHED for uncached pages.

### 6.1.21 KMFreePages

VOID \_\_stdcall KMFreePages( PVOID pvAddress, DWORD dwNumPages, DWORD dwFlags );

pvAddress – linear address of the pages to free.

dwNumPages – number of pages to free

dwFlags – currently unused

The pvAddress and dwNumPages must be the values that were returned by a call to KMAllocPages. Freeing of subranges is not supported.

### 6.1.22 KMGetCmndBufferInfo

DWORD \_\_stdcall KMGetCmndBufferInfo(PDEVICE\_CTLBLK psDeviceCtlBlk, PCMND\_BUF\_INFO psGetCmndInfo);

Retrieves information about the circular command buffer for a specified device. The returned structure is defined as :

typedef struct \_CMNDBUFINFO\_

{

PBYTE	pbyCmndBufLinBase;	-- linear base of command buffer
DWORD	dwCmndBufPhysBase;	-- physical base of command buffer
PDWORD	pdwAccess;	-- linear address of access field
PDWORD	pdwWOff;	-- linear address of write offset cursor
PDWORD	pdwROff;	-- linear address of read offset cursor

```

PDWORD    pdwKick;                -- linear address of kicker register
DWORD     dwCBInHost;             -- Not Used
} CMND_BUF_INFO, *PCMND_BUF_INFO;

```

These are all outputs, except for dwCBInHost, which is not currently used.

### 6.1.23 KMSetCmndBufferInfo

```

DWORD __stdcall KMSetCmndBufferInfo(PDEVICE_CTLBLK psDeviceCtlBlk,
PSET_CMND_BUF_INFO psSetCmndInfo);

```

This function takes a SET\_CMND\_BUF\_INFO structure of the form

```

typedef struct _SET_CMND_BUF_INFO_
{
    PDWORD    pdwAccess;    -- linear address of access field for this CCB
    PDWORD    pdwWOff;     -- linear address of write offset cursor
    PDWORD    pdwROff;     -- linear address of read offset cursor
} SET_CMND_BUF_INFO, *PSET_CMND_BUF_INFO;

```

The intent is for the display driver to set these values early on.

### 6.1.24 KMAllocHostMem

```

DWORD __stdcall KMAllocHostMem(DWORD dwFlags, DWORD dwSize, PVOID
*ppvLinAddr, DWORD *pdwPhysAddr);

```

Allocates memory on one of the host heaps. The heap can be chosen by setting the dwFlags word to be one of :

```

KM_HOST_PAGEABLE_HEAP
KM_HOST_NON_PAGEABLE_HEAP
KM_HOST_NON_CACHED_HEAP

```

The function will return an error code if it fails, or KM\_OK if it succeeds.

### 6.1.25 KMFreeHostMem

DWORD \_\_stdcall KMFreeHostMem(DWORD dwFlags, PVOID pvLinAddr);

Frees a block of memory in the specified heap, given the linear address of the block.

dwFlags must be set to the heap from which the block was originally allocated.

### 6.1.26 KMWriteOBC

DWORD \_\_stdcall \*KMWriteOBC(PDEVICE\_CTLBLK psDCB, DWORD dwCount, PWORD pwReg, PWORD pwData);

This function is not applicable to PMX2

Writes several values to a PMX devices' out-of-band control registers.

psDCB           – handle of the device to write to  
dwCount       – number of registers to write to  
pwReg          – array of 16-bit words containing the numbers of the registers to write to  
pwData        – array of 16-bit words containing the values to write into the corresponding registers

### 6.1.27 KMReadOBC

DWORD \_\_stdcall KMReadOBC(PDEVICE\_CTLBLK psDCB, DWORD dwCount, PWORD pwReg, PWORD pwData);

This function is not applicable to PMX2

Reads several values from a PMX devices' out-of-band control registers.

psDCB           – handle of the device to read from  
dwCount       – number of registers to read from  
pwReg          – array of 16-bit words containing the numbers of the registers to read from  
pwData        – array of 16-bit words containing the read from the corresponding registers

### 6.1.28 KMWriteOBCAI

DWORD \_\_stdcall KMWriteOBCAI(PDEVICE\_CTLBLK psDCB, DWORD dwCount, DWORD wIReg, WORD wIData, WORD wDReg, PWORD pwData);

This function is not applicable to PMX2

Writes dwCount auto incrementing data values to OBC via SPAM.

psDCB -- identifier for this device  
dwCount -- count of data values to write to data reg  
wIReg -- auto incrementing index register address  
wIData -- seed value for ai index reg.  
wDReg -- data register address  
pwData -- (16-bit) word array of data to write to data register.

### 6.1.29 KMReadOBCAI

DWORD \_\_stdcall KMReadOBCAI(PDEVICE\_CTLBLK psDCB, DWORD dwCount, WORD wIReg, WORD wIData, WORD wDReg, PWORD pwData);

This function is not applicable to PMX2

Reads dwCount auto incremented data values from OBC via SPAM.

psDCB -- identifier for this device  
dwCount -- count of regs to read  
wIReg -- auto incrementing index register address  
wIData -- seed value for ai index reg.  
wDReg -- data register address  
pwData -- word array of data to read from data register.

### 6.1.30 KMReadOBCNB

DWORD \_\_stdcall KMReadOBCNB(PDEVICE\_CTLBLK psDCB, DWORD dwCount, PWORD pwReg, PWORD pwData);

This function is not applicable to PMX2



As for KMReadOBC, except that if another read is being carried out on the same device, this returns straightaway with the error KM\_ERROR\_RESOURCE\_INUSE.

It is intended to be called from inside an interrupt routine.

### 6.1.31 KMGetCardInfo

DWORD \_\_stdcall KMGetCardInfo(PDEVICE\_CTLBLK psDCB, PKMCARDINFO psInfo);

Fills out the passed in structure with various bits of information about the specified device.

The structures involved are:

typedef struct \_KMCARDINFO\_

```
{
    DWORD          dwStructSize;
    DWORD          dwErrorCode;
    DEV_PCI        sPCI;           -- PCI config information
    DEV_LOCATION    sMem;         -- Memory information
    PDEVICE_CTLBLK psDeviceCtlBlk;
}
```

KMCARDINFO, \*PKMCARDINFO;

typedef struct \_DEV\_PCI\_

```
{
    DWORD          dwBus;         --Bus for this device
    DWORD          dwSlot;        --Slot for this device
    DWORD          dwFunc;        --Function for this device
    DWORD          dwHWRev;       --Hardware revision (1 = PMX1C, 2 = PMXLC)
    DWORD          dwIRQ;
    DWORD          dwDeviceID;
    DWORD          dwVendorID;
    DWORD          dwSubSysID;
    DWORD          dwSubSysVendorID;
}
```



```
    DWORD    dwBIOSRev;
    DWORD    dwBoardRev;        --This is defunct - don't use
    DWORD    dwIsAGPCard;       --1 if this is an AGP card
}
DEV_PCI;

typedef struct _DEV_LOCATION_
{
    DWORD    dwFBPhysicalBase;    -- Physical base of frame buffer
    DWORD    dwFBLinBase;        -- Linear base of frame buffer
    DWORD    dwFBMemorySize;     -- Size of frame buffer memory
    DWORD    dwSysRegsPhysBase;  -- Physical base of registers
    DWORD    dwSysRegsLinBase;   -- Linear base of registers
    DWORD    dwComsRegsPhysBase;
    DWORD    dwComsRegsLinBase;
    DWORD    dwROMPhysicalBase;   -- Physical base of ROM
    DWORD    dwROMLinBase;       -- Linear base of ROM
    DWORD    dwROMSize;          -- Size of ROM (bytes)
    DWORD    dwATLUTRegionBase;  -- Deprecated. Do not use.
} DEV_LOCATION;
```

### 6.1.32 KMAAllocHostCacheCoherentHeap

```
DWORD __stdcall KMAAllocHostCacheCoherentHeap(PDEVICE_CTLBLK psDCB,
DWORD dwSize, DWORD dwAlignment, PVOID *ppvLinAddr, DWORD
*pdwPMXAddr);
```

Allocates host memory which is guaranteed to be cache coherent with the PMX device. If the PMX device reads from this region, then it is guaranteed to flush the host's caches in order to provide the data. Note that general reads from host memory will not usually be cache coherent (i.e. they won't take any notice of what is in the processors cache).



### 6.1.33 KMFreeHostCacheCoherentHeap

```
DWORD __stdcall *KMFreeHostCacheCoherentHeap)(PDEVICE_CTLBLK psDCB,  
PVOID pvLinAddr);
```

Frees memory allocated in the cache coherent heap of the specified device.

### 6.1.34 KMPower

```
DWORD __stdcall KMPower(PDEVICE_CTLBLK psDCB, DWORD dwFlags, PDWORD  
pdwState);
```

Notifies the Kernel manager of a power mode change on a particular device, or reads the current power mode.

dwFlags -- indicates whether we're setting or getting a state

KM\_POW\_GETSTATE

KM\_POW\_SETSTATE

pdwState -- pointer to a DWORD to receive or supply the state value

this value will be one of:

KM\_POW\_STATE0 -- Running full speed

KM\_POW\_STATE1 -- Not currently supported

KM\_POW\_STATE2 -- Not currently supported

KM\_POW\_STATE3 -- Device power off state

### 6.1.35 KMCorePLL

```
DWORD __stdcall KMCorePLL(PDEVICE_CTLBLK psDCB, DWORD dwFlags,  
PDWORD pdwFreq);
```

Sets or gets the core PLL frequency of the specified device.



dwFlags is one of:

KM\_CORE\_PLL\_SET

KM\_CORE\_PLL\_GET

pdwFreq points to a DWORD which receives or supplies the the clock frequency in MHz.

### 6.1.36 KMState

DWORD \_\_stdcall KMState(PDEVICE\_CTLBLK psDCB, DWORD dwState);

Notifies the Kernel Manager of a state change. This will typically call a callback on components which have registered with the Kernel Manager to receive state change notifications. (see KMRegisterStateChangeCallback below).

dwState is one of:

PMX\_PRE\_WINDOWS\_MODE\_CHANGE

PMX\_PRE\_HIREZ\_TO\_VGA

PMX\_POST\_HIREZ\_TO\_VGA

PMX\_PRE\_VGA\_TO\_HIREZ

PMX\_POST\_VGA\_TO\_HIREZ

PMX\_PRE\_XY\_REZ\_CHANGE

PMX\_POST\_XY\_REZ\_CHANGE

PMX\_PRE\_Z\_REZ\_CHANGE

PMX\_POST\_Z\_REZ\_CHANGE

### 6.1.37 KMGetAPITable

DWORD \_\_stdcall KMGetAPITable( PCODE\_CTLBLK psCodeCtlBlk, PDWORD \*ppdwAPITable );

Fills out the PDWORD pointed to by ppdwAPITable with the linear address of the API table for the given code fragment. No device handle is necessary, as the code control block implicitly specifies a device anyway.



### 6.1.38 KMRegisterStateChangeCallback

```
DWORD __stdcall KMRegisterStateChangeCallback( PDEVICE_CTLBLK psDeviceCtlBlk,
PSTATE_CALLBACK pfnCallback );
```

Registers a callback with the kernel manager to be called when the power mode changes, or when we receive a DOS box pre or post mode change message.

The callback function should be of the form:

```
Fn( PDEVICE_CTLBLK psDeviceCtlBlk, DWORD dwStateFlags, DWORD
dwContextLossSeverityFlags )
```

Be sure to use the `__stdcall` declaration in defining your callback. Casting to a `__stdcall` type is not sufficient.

### 6.1.39 KMDeRegisterStateChangeCallback

```
DWORD __stdcall KMDeRegisterStateChangeCallback( PDEVICE_CTLBLK
psDeviceCtlBlk, PSTATE_CALLBACK pfnCallback )
```

Removes a state change callback from the list of handlers

### 6.1.40 KMEnableDevice

```
DWORD __stdcall KMEnableDevice( PDEVICE_CTLBLK psDCB )
```

Initialises the heaps, etc. and starts the chip running in Windows (non-VGA) mode for a particular device. No other device-specific call may be made before this. (KMMMapOSDevCookie is OK, obviously).

## 7. DEVICEIOCONTROLINTERFACE REFERENCE

This interface allows ring3 applications to access the Kernel Manager, by using an IOCTL (win9x only). It is used for testing.

The format of these calls is:

```
DeviceIoControl( hVxd, KM_FUNCTION_NUMBER, &sInOut, sizeof(
KM_FUNCTION_STRUCTURE ), &sInOut, sizeof(
KM_FUNCTION_STRUCTURE ), &dwBytesReturned, NULL );
```

where

hVxd is a handle to the Kernel Manager VxD

KM\_FUNCTION\_NUMBER is a number indicating which function to call (specified in kmapi.h)

KM\_FUNCTION\_STRUCTURE is the type of structure demanded by the function being called. This holds all input/output parameters.

sInOut is the structure (of type KM\_FUNCTION\_STRUCTURE) containing the parameters of the call. This contains a dwErrorCode field, so that all functions can indicate an error code.

The following functions are currently supported, in almost every case they behave exactly as the corresponding KM function. Only the exceptions are described in detail.

### 7.1.1 KM\_API\_LOAD\_CODEFRAGMENT

Uses CODE\_DETAILS structure as the interface

typedef struct \_CODE\_DETAILS\_

```
{
    DWORD          dwStructSize;      (Do not use)
    DWORD          dwErrorCode;       (Output)
    PDWORD         pdwInfoBuffer;     (Input)
    DWORD          dwLoadFlags;       (Input – should be 0)
    DWORD          dwThread;          (Input)
    PCODE_CTLBLK   psCodeCtlBlk;      (Output)
    PDEVICE_CTLBLK psDeviceCtlBlk;    (Input)
} CODE_DETAILS, *PCODE_DETAILS;
```

### 7.1.2 KM\_API\_UNLOAD\_CODEFRAGMENT

Unloads a code fragment. Uses a CODE\_DETAILS structure as the interface.

### 7.1.3 KM\_API\_FIND\_THREAD\_WITH\_CAPS

Uses a FIND\_THREAD structure as the interface

### 7.1.4 KM\_API\_GET\_CODECTLBLOCKPTR

Uses the following structure as the interface:

```
typedef struct _GET_CTLBLKPTR_
{
    DWORD          dwStructSize;      (Do not use)
    DWORD          dwErrorCode;       (Output)
    PSZ            pszFragName;       (Input)
    PDEVICE_CTLBLK psDeviceCtlBlk;    (Input)
    PCODE_CTLBLK   psCodeCtl;         (Output)
} GET_CTLBLKPTR, *PGET_CTLBLKPTR;
```

### 7.1.5 KM\_API\_GET\_SYMBOL\_ADDRESS

Uses the following structure as the interface:

```
typedef struct _GET_SYMBOL_ADDR_
{
    DWORD          dwStructSize;
    DWORD          dwErrorCode;
    PSZ            pszName;
    PCODE_CTLBLK   psCodeCtl;
    DWORD          dwHighAddress;
    PVOID          pvHostAddress;
} GET_SYMBOL_ADDR, *PGET_SYMBOL_ADDR;
```

This is different from the ring0 function in that here we have a single call that returns both linear and physical addresses of the specified symbol.

### 7.1.6 KM\_API\_GET\_API\_TABLE

Uses the following structure as the interface:

```
typedef struct _GET_API_TABLE_
{
    DWORD          dwStructSize;
    DWORD          dwErrorCode;
```



```
PCODE_CTLBLK    psCodeCtlBlk;
PDWORD          pdwAPITable;
} GET_API_TABLE, *PGET_API_TABLE;
```

### 7.1.7 KM\_API\_MAP\_OSDEVCOOKIE\_TO\_DEVICE

Uses the following structure as the interface:

```
typedef struct _MAP_OSDEVCOOKIE_
{
    DWORD          dwStructSize;
    DWORD          dwErrorCode;
    DWORD          sOSDevCookie;
    PDEVICE_CTLBLK psDeviceCtlBlk;
} MAP_OSDEVCOOKIE, *PMAP_OSDEVCOOKIE;
```

### 7.1.8 KM\_API\_GET\_CARD\_INFO

Uses the following structure as the interface:

```
typedef struct _KMCARDINFO_
{
    DWORD          dwStructSize;
    DWORD          dwErrorCode;
    DEV_PCI        sPCI;
    DEV_LOCATION   sMem;
    PDEVICE_CTLBLK psDeviceCtlBlk;
}
KMCARDINFO, *PKMCARDINFO;
```

See KMGetCardInfo for more information on the DEV\_PCI and DEV\_LOCATION structures.

### 7.1.9 KM\_API\_LOCK\_MOVEABLE\_BUFFER

Uses the following structure as the interface:

```
typedef struct _MOVE_LOCK_UNLOCK_
{

```



```

        DWORD    dwStructSize;
        DWORD    dwErrorCode;
        PVOID     pvHandle;
        PVOID     pvHostAddress;
        DWORD     dwHighAddress;
    } MOVE_LOCK_UNLOCK, *PMOVE_LOCK_UNLOCK;

```

This will lock the block with the specified handle, and write the linear and physical addresses of the locked block into pvHostAddress and dwHighAddress.

#### 7.1.10 KM\_API\_UNLOCK\_MOVEABLE\_BUFFER

This uses the MOVE\_LOCK\_UNLOCK structure, but it ignores the pvHostAddress and dwHighAddress fields.

It unlocks the moveable block with the specified pvHandle.

#### 7.1.11 KM\_API\_MAP\_LIN\_TO\_PHYS

This uses the following structure:

```

typedef struct _MAP_LIN_TO_PHYS_
{
    DWORD    dwStructSize;
    DWORD    dwErrorCode;
    DWORD    dwLinAddr;  (Input)
    DWORD    dwPhysAddr; (Output)
} MAP_LIN_TO_PHYS, *PMAP_LIN_TO_PHYS;

```

Given a linear address dwLinAddr, this finds the corresponding physical address, and returns it in dwPhysAddr.

#### 7.1.12 KM\_API\_ALLOC\_STATIC\_FB\_MEMORY

This uses the following structure:

```

typedef struct _STATIC_ALLOC_
{

```



```
    DWORD        dwStructSize;
    DWORD        dwErrorCode;
    DWORD        dwSize;
    DWORD        dwLocation;
    PVOID        pvHostAddress;
    DWORD        dwHighAddress;
    PDEVICE_CTLBLK psDeviceCtlBlk;
} STATIC_ALLOC, *PSTATIC_ALLOC;

dwLocation should be set to the same set of flags as in KMAllocStaticFBMem
```

### 7.1.13 KM\_API\_FREE\_STATIC\_FB\_MEMORY

This uses the structure below:

```
typedef struct _STATIC_FREE_
{
    DWORD        dwStructSize;
    DWORD        dwErrorCode;
    PVOID        pvAddr;
    DWORD        dwLocation;
    PDEVICE_CTLBLK psDeviceCtlBlk;
} STATIC_FREE, *PSTATIC_FREE;
```

Again, dwLocation must be one of the region flags used in KMAllocStaticFBMem.

pvAddr is the linear address returned by KM\_API\_ALLOC\_STATIC\_FB\_MEM

### 7.1.14 KM\_API\_GET\_AVAILABLE\_MEMORY

This uses an AVAIL\_MEM structure as the interface.

### 7.1.15 KM\_API\_ALLOC\_MOVE\_FB\_MEMORY

This takes a MOVE\_ALLOC structure:

```
typedef struct _MOVE_ALLOC_
{
```



```
    DWORD dwStructSize;
    DWORD dwErrorCode;
    DWORD dwSize;
    DWORD dwLocation;
    PVOID pvHandle;
    PDEVICE_CTLBLK psDeviceCtlBlk;
} MOVE_ALLOC, *PMOVE_ALLOC;
```

pvHandle is a handle to the (unlocked) moveable block.

dwLocation is one of the standard set of frame buffer allocation flags.

#### 7.1.16 KM\_API\_FREE\_MOV\_FB\_MEMORY

This uses the following structure as its interface

```
typedef struct _MOVE_FREE {
{
    DWORD dwStructSize;
    DWORD dwErrorCode;
    PDEVICE_CTLBLK psDeviceCtlBlk;
    DWORD dwLocation;
    PVOID pvHandle;
} MOVE_FREE, *PMOVE_FREE;
```

psDeviceCtlBlk and dwLocation should be set up as they were for the allocation of the block.  
pvHandle should be the value returned from KM\_API\_ALLOC\_MOV\_FB\_MEMORY

#### 7.1.17 KM\_API\_ALLOC\_HOST\_MEMORY

This uses a STATIC\_ALLOC structure in its interface, except that the dwLocation field should be set to one of the host heap location flags:

```
KM_HOST_PAGEABLE_HEAP
KM_HOST_NON_PAGEABLE_HEAP
KM_HOST_NON_CACHED_HEAP
```



### 7.1.18 KM\_API\_FREE\_HOST\_MEMORY

This takes a `STATIC_FREE` structure, with the `dwLocation` field being one of the host heap location flags.

### 7.1.19 KM\_GET\_SERVICE\_TABLE

This takes the following structure, which it fills out with the relevant data

typedef struct tagKMSERVICE

```
{
    DWORD                dwNumServices;
    PKM_SERVICE_TABLE    psService;
}
```

KMSERVICE, \*PKMSERVICE;

the definition of a `KM_SERVICE_TABLE` is given in `kmapi.h` – it includes all the ring0 functions.

### 7.1.20 KM\_API\_ALLOC\_CMDIF

This uses the following structure:

typedef struct \_ALLOC\_CMDIF\_

```
{
    DWORD                dwStructSize;
    DWORD                dwErrorCode;
    DWORD                dwThread;    (Input)
    DWORD                dwFlags;    (must be zero)
    PCOMMANDIF           psCommandIF;
    DWORD                dwHPtrCommandIF;
    PDEVICE_CTLBLK       psDeviceCtlBlk;
} ALLOC_CMDIF, *PALLOC_CMDIF;
```

Given a thread number and a device handle, this allocates a command interface for that thread. It returns the linear address of the allocated command interface (in `psCommandIF`) as well as its physical address (in `dwHPtrCommandIF`)

### 7.1.21 KM\_API\_FREE\_COMIF

This takes the same ALLOC\_CMDIF structure as KM\_API\_ALLOC\_COMIF. It ignores all fields except the linear and physical address of the command interface

### 7.1.22 KM\_API\_SET\_CMND\_BUF\_INFO

This uses the following structure (CCB refers to the circular command buffer)

```
typedef struct _DIOC_CMND_BUF_INFO_
{
    PDEVICE_CTLBLK psDeviceCtlBlk;
    PBYTE          pbyCmndBufLinBase;    -- (Ignored)
    DWORD          dwCmndBufPhysBase;    -- (Ignored)
    PDWORD         pdwAccess;             -- (Input) Linear address of access flag
    PDWORD         pdwWOff;               -- (Input) Linear address of Woff cursor
    PDWORD         pdwROff;               -- (Input) Linear address of Roff cursor
    PDWORD         pdwKick;               -- (Output) Address of kicker
    DWORD          dwCBInHost;            -- Not used
} DIOC_CMND_BUF_INFO, *PDIOC_CMND_BUF_INFO;
```

### 7.1.23 KM\_API\_GET\_CMND\_BUF\_INFO

This uses the same DIOC\_CMND\_BUF\_INFO structure as KM\_API\_SET\_CMND\_BUF\_INFO, although it is interpreted differently.

```
typedef struct _DIOC_CMND_BUF_INFO_
{
    PDEVICE_CTLBLK psDeviceCtlBlk;
    PBYTE          pbyCmndBufLinBase;    -- (Output) linear base of CCB
    DWORD          dwCmndBufPhysBase;    -- (Output) physical base of CCB
    PDWORD         pdwAccess;             --(Output) Linear address of access flag
    PDWORD         pdwWOff;               --(Output) Linear address of Woff cursor
    PDWORD         pdwROff;               --(Output) Linear address of Roff cursor
    PDWORD         pdwKick;               --(Output) Address of kicker
    DWORD          dwCBInHost;            --Not used
} DIOC_CMND_BUF_INFO, *PDIOC_CMND_BUF_INFO;
```

## 8. MACROS

### 8.1.1 FENCED\_WRITE( dwAddr, dwData )

Performs a fenced write of the given DWORD to the given (linear) address. The means that the processor will flush all outstanding writes in its write buffer before starting this fenced write. It is therefore used to enforce in-order writes to memory on systems with write-combining.

### 8.1.2 CMND\_BUFFER\_SIZE

Defines the size of the circular command buffer, in bytes.

### 8.1.3 CMND\_BUFFER\_OVERFLOW

Defines the size of the circular command buffer overflow area, in bytes.

### 8.1.4 GET\_PCCB\_SPACE(Woff,Roff)

Given (de-referenced) Woff and Roff cursors, this will return the amount of space (in bytes) currently in the circular command buffer.

### 8.1.5 UPDATE\_PCCB\_WOFF(USize,Woff)

Given a size in bytes, and a (de-referenced) Woff cursor, this will modify the Woff to point to the next free item in the circular command buffer. Usize must be a multiple of the current primary core cache line size (currently 64 bytes).

### 8.1.6 CalcParamSize

```
__inline DWORD CalcParamSizeCCB(DWORD dwParamSize);
```

Returns the input ParamSize, rounded up to a multiple of 64.

### 8.1.7 ReleaseCCB

```
__inline void ReleaseCCB(PCMND_BUF_INFO psPCCB);
```

Frees the CCB by zeroing its dwAccess field

### 8.1.8 GetCCB

Waits for CCB access rights and checks for available space in CCB for blit param structure

```
__inline PVOID GetCCB(PCMND_BUF_INFO psPCCB, DWORD dwParamSize);
```

psPCCB                    -- pointer to CCB information struct

dwParamSize            -- size of blit data structure

Returns the new address of next free slot to write to if successful, else returns 0

## 9. SIMULATION BUILDS OF THE KERNEL MANAGER

For running software primary core / secondary core simulations, a special build of the kernel manager (build flag SIM=1) is required. This build will then obtain the following information from the registry on startup:

\HKEY\_LOCAL\_MACHINE\SOFTWARE\VideoLogic\PMXSIM

FreeCommandIFs	String	if 0 then command interfaces will never be freed by KMFreeComIFs. This setting is used for the generation of hardware simulation data.
SimCardMemorySize	String	Amount of memory on the simulated card
SimHardwareVersion	String	1 for PMX1 2 for LC, 3 for PMX2
SimPhysicalBase	String	Physical Base of the simulated frame buffer
SimTextureMemoryOffset	String	Offset into the frame buffer at which to start texture memory
FreeCommandIFs	String	Indicates whether we should free command interfaces when asked to do so, or allow them to build up in memory (used for hardware sims). For normal operation this will be set to "0x00000000"

Values stored as strings in the registry are interpreted as hex strings if you prefix them with "0x". In fact no other interpretation is currently permitted.

